# 6.9 Cây tìm kiếm nhị phân (BST)

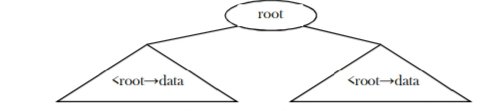
## Tại sao cây tìm kiếm nhị phân?

## Trong các phần trước, chúng tôi đã thảo luận về cách biểu diễn cây, không áp đặt hạn chế nào đối với dữ liệu nút, dẫn đến độ phức tạp tìm kiếm là O(n) trong trường hợp xấu nhất.

## Trong phần này, chúng ta sẽ thảo luận về biến thể khác của cây nhị phân: Cây tìm kiếm nhị phân (BST). Mục đích chính của BST là tìm kiếm, áp đặt hạn chế về loại dữ liệu mà một nút có thể chứa. Kết quả là giảm độ phức tạp tìm kiếm trung bình trong trường hợp xấu nhất xuống O(logn).

## Thuộc tính cây tìm kiếm nhị phân

Trong cây tìm kiếm nhị phân, thuộc tính quan trọng nhất là mọi phần tử trong cây con bên trái đều phải nhỏ hơn dữ liệu gốc và mọi phần tử trong cây con bên phải phải lớn hơn dữ liệu gốc. Điều này được gọi là thuộc tính cây tìm kiếm nhị phân, và nó phải được thỏa mãn tại mỗi nút trong cây. Điều này có nghĩa là:

1. Cây con bên trái của một nút chỉ chứa các nút có giá trị (khóa) nhỏ hơn giá trị của nút đó.
2. Cây con bên phải của một nút chỉ chứa các nút có giá trị lớn hơn giá trị của nút đó.
3. Cả cây con bên trái và cây con bên phải đều phải là cây tìm kiếm nhị phân.

Các thao tác chính trên cây tìm kiếm nhị phân bao gồm tìm/tìm tối thiểu/tìm tối đa, chèn phần tử, và xóa phần tử. Các hoạt động phụ trợ bao gồm kiểm tra xem cây là cây tìm kiếm nhị phân, tìm phần tử nhỏ thứ k, và sắp xếp các phần tử.

Trong quá trình giải quyết bài toán trên cây tìm kiếm nhị phân, việc duyệt theo thứ tự sẽ tạo ra một danh sách được sắp xếp. Quy trình này thường bắt đầu từ cây con bên trái, tiếp theo là xử lý dữ liệu gốc và sau cùng là cây con bên phải.

Cây tìm kiếm nhị phân tận dụng thuộc tính cây tìm kiếm để giảm thời gian tìm kiếm, điều này giúp nó hiệu quả hơn so với cây nhị phân thông thường. Thao tác tìm kiếm trên cây tìm kiếm nhị phân chỉ xem xét một trong hai cây con để tìm kiếm phần tử, giảm độ phức tạp so với cây nhị phân thông thường.

Các thao tác cơ bản trên cây tìm kiếm nhị phân, như chèn phần tử, xóa phần tử và tìm kiếm phần tử, mất thời gian tỷ lệ thuận với chiều cao của cây. Trong trường hợp tốt nhất (cây hoàn chỉnh), độ phức tạp là O(logn), nhưng trong trường hợp xấu nhất (cây xiên), độ phức tạp có thể là O(n).

## Tìm một phần tử trong cây tìm kiếm nhị phân

Hoạt động tìm kiếm rất đơn giản trong BST. Bắt đầu với gốc và tiếp tục di chuyển sang trái hoặc phải bằng thuộc tính BST. Nếu dữ liệu chúng tôi đang

tìm kiếm giống với dữ liệu nút thì chúng tôi trả về nút hiện tại. Nếu dữ liệu chúng ta đang tìm kiếm nhỏ hơn dữ liệu nút thì tìm kiếm cây con bên trái

của nút hiện tại; mặt khác tìm kiếm cây con bên phải của nút hiện tại. Nếu dữ liệu không có, chúng ta sẽ có một liên kết rỗng.

class BinarySearchTreeNode:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

def find(root, data):

    if root is None:

        return None

    if data < root.data:

        return find(root.left, data)

    elif data > root.data:

        return find(root.right, data)

    return root

Độ phức tạp về thời gian: O(n), trong trường hợp xấu nhất (khi cây tìm kiếm nhị phân đã cho là cây nghiêng). Độ phức tạp của không gian: O(n), đối với ngăn xếp đệ quy.

Phiên bản không đệ quy của thuật toán trên có thể được đưa ra dưới dạng:

class BinarySearchTreeNode:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

def find(root, data):

    while root is not None:

        if data == root.data:

            return root

        elif data > root.data:

            root = root.right

        else:

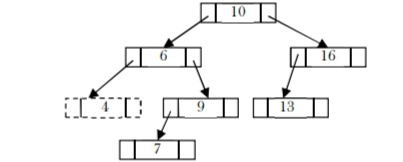
            root = root.left

    return None

Độ phức tạp về thời gian: O(n). Độ phức tạp của không gian: O(1).

## Tìm phần tử tối thiểu trong cây tìm kiếm nhị phân

Trong BST, phần tử tối thiểu là nút ngoài cùng bên trái, không có nút con bên trái. Trong BST bên dưới, phần tử tối thiểu là 4.



class BinarySearchTreeNode:

    def \_\_init\_\_(self, left=None, right=None):

        self.left = left

        self.right = right

def find\_max(root):

    if root is None:

        return None

    elif root.right is None:

        return root

    else:

        return find\_max(root.right)

BST là cây nghiêng trái). Độ phức tạp của không gian: O(n), đối với ngăn xếp đệ quy.

Độ phức tạp về thời gian: O(n), trong trường hợp xấu nhất (khi phiên bản không đệ quy của thuật toán trên có thể được đưa ra dưới dạng:

class BinarySearchTreeNode:

    def \_\_init\_\_(self, left=None, right=None):

        self.left = left

        self.right = right

def find\_max(root):

    if root is None:

        return None

    while root.right is not None:

        root = root.right

    return root

Độ phức tạp về thời gian: O(n). Độ phức tạp của không gian: O(1).

## Chèn một phần tử từ cây tìm kiếm nhị phân

Để chèn dữ liệu vào cây tìm kiếm nhị phân, quy trình bắt đầu bằng việc tìm vị trí phù hợp cho phần tử mới. Điều này được thực hiện bằng cách sử dụng cơ chế tương tự như thao tác tìm. Trong quá trình tìm kiếm, nếu dữ liệu đã tồn tại ở một nút, thì ta chỉ cần bỏ qua và kết thúc tìm kiếm. Ngược lại, chèn dữ liệu vào vị trí cuối cùng trên đường đi.

Ví dụ, nếu ta muốn chèn phần tử 5 vào cây tìm kiếm nhị phân, ta duyệt cây bắt đầu từ gốc. Khi ta đến nút có khóa là 4, ta cần sang phải vì 5 lớn hơn 4, nhưng không có cây con ở đó. Điều này cho biết rằng 5 không tồn tại trong cây, và đây là vị trí chính xác để chèn phần tử mới.

class BinarySearchTreeNode:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

def insert(root, key):

    if root is None:

        return BinarySearchTreeNode(key)

    else:

        if key < root.data:

            root.left = insert(root.left, key)

        elif key > root.data:

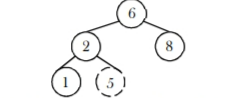
            root.right = insert(root.right, key)

    return root

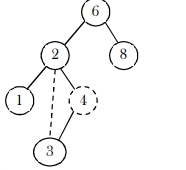
## Xóa một phần tử khỏi cây tìm kiếm nhị phân

Thao tác xóa phức tạp hơn các thao tác khác. Điều này là do phần tử cần xóa có thể không phải là nút lá. Trong thao tác này cũng vậy, trước tiên chúng ta cần tìm vị trí của phần tử mà chúng ta muốn xóa. Khi chúng tôi đã tìm thấy nút cần xóa, hãy xem xét Nếu phần tử cần xóa là nút lá: trả về NULL cho nút cha của nó. Điều đó có nghĩa là làm cho con trỏ con tương ứng là NULL. Trong cây bên dưới để xóa 5, đặt NULL thành nút cha 2 của nó.

các trường hợp sau:



Nếu phần tử cần xóa có một phần tử con: Trong trường hợp này chúng ta chỉ cần gửi phần tử con của nút hiện tại tới nút cha của nó. Trong cây bên dưới, để xóa 4, 4 cây con bên trái được đặt thành nút cha 2 của nó.



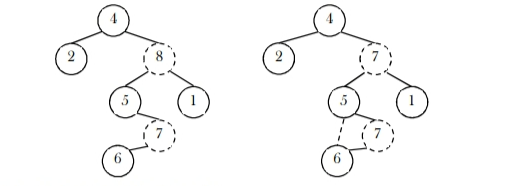
Nếu phần tử cần xóa có cả hai phần tử con: Chiến lược chung là thay thế khóa của nút này bằng phần tử lớn nhất của cây con bên

trái và xóa đệ quy nút đó (hiện đang trống). Nút lớn nhất trong cây con bên trái không thể có nút con bên phải, vì vậy việc xóa thứ

hai là một việc dễ dàng. Ví dụ, chúng ta hãy xem xét cây sau đây. Trong cây bên dưới, để xóa 8, nó là con bên phải của gốc.

Giá trị khóa là 8. Nó được thay thế bằng khóa lớn nhất trong cây con bên trái của nó (7), và sau đó nút đó sẽ bị xóa như trước (trường

hợp thứ hai).



class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

def find\_max(node):

    current = node

    while current.right is not None:

        current = current.right

    return current

def delete(root, data):

    if root is None:

        print("Element not there in tree")

        return root

    if data < root.data:

        root.left = delete(root.left, data)

    elif data > root.data:

        root.right = delete(root.right, data)

    else:

        if root.left is not None and root.right is not None:

            temp = find\_max(root.left)

            root.data = temp.data

            root.left = delete(root.left, temp.data)

        else:

            if root.left is None:

                root = root.right

            elif root.right is None:

                root = root.left

    return root

Độ phức tạp về thời gian: O(n). Độ phức tạp về không gian: O(n) cho ngăn xếp đệ quy. Đối với phiên bản lặp lại, độ phức tạp về không gian là O(1).